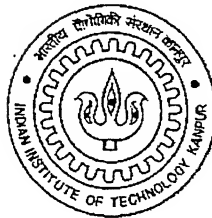# A Debugger for

Distributed Systems

A Thesis Submitted in Partial Fulfillment of

the Requirements for the Degree of

Master of Technology

by

Pratik Y. Mehta

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

July, 2005

Certificate

This is to certify that the work contained in the thesis entitled "A Debugger for Distributed Systems" by Pratik Y. Mehta has been carried out under my supervision and it has not been submitted elsewhere for a degree.
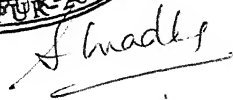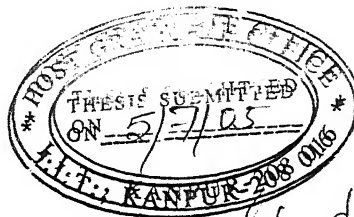
July, 2005

(Dr. Sanjeev K. Aggarwal)

Department of Computer Science and Engineering,

Indian Institute of Technology, Kanpur

Abstract

Distributed execution is attractive and sometimes the only approach for improving quality attributes of software like performance, reliability and scalability. As debugging distributed executions is a daunting task, simplification of distributed debugging process is required. We show simple semantic handles in the distributed executions. These semantic handles can be used for narrowing the gap between the model of execution platform and the anticipated model of software behavior. They provide hints for good design of a language used for programming distributed systems.

We develop an infrastructure for testing distributed debugging protocols. It can be used to test new protocols and methodologies for distributed debugging. We develop a sample protocol for distributed applications designed using SPMD model.

Acknowledgments

To begin with, I would like to express gratitude for thesis adviser Dr. Sanjeev K. Aggarwal. Apart from playing role of an exemplar in academic research, He organized the course of thesis with great perseverance. As a result, balance of novel explorations and focus towards goals was preserved.

I would also like to thank the faculty of Computer Science and Engineering Department who directly or indirectly influenced the work through courses and TA duties I have done along with informal discussions. The office staff also deserves a word of appreciation for the efficiency in administration.

Thanks to friends and well wishers for going out of the way to extend friendly gestures and contributing in creation of live environment.

I am ever indebted to the deity, my parents and family for all small and great ways in which their blessing are bestowed.

On a fine morning of a bright day,

Thought of making spaghetti came my way,

And readied was the pan to cook it fine,

Invited all good people to join and dine,

Pasta looked good and smelled even better,

And proud was I to announce its maker,

"A bug" screamed the lady sitting nearby,

Answered I, looks it like that; when fry,

To find that really strange looked that shape,

Too embarrassed to admit, standing awhape,

Tried to catch and grab it as deeper it went,

Broken into more pieces till I counted to cent,

Started cooking afresh, after tribute was paid,

Action plans to avoid further bugs were laid;

"'Debugging' it shall be called" then I declared,

To counter a bug thus a debugger was prepared.

Contents

List of Figures

1 Introduction

1.1 Motivation

Development tools for distributed systems have to counter new challenges in addition to challenges for development tools for non-distributed systems. Distributed computing theory attempts to build general models that can be later tailored to specific assumptions. However, the models of computing have wide range of variety in deployment. Due to this, there is much room for evaluating the trade-off of performance and the abstractions. Some basic structures defined by protocols or topologies like remote procedure calls exist. An attempt to fully abstract out the concerns of distributed systems results in a weaker model of the system. This is not suitable choice for high performance computing applications due to bundled overhead and penalty of weak model. Hence, it is required to understand structures that provide semantic handles in distributed executions. It is not always necessary to fully abstract out distributed nature of

computation. A semantic handle is an abstraction of program behavior which closely relates to a higher level concept.

1.2   Status of distributed debugging

The problem of distributed debugging is subjected to extensive research in past few decades. An overview of work in field can be found in [GVS94, MH89]. A more recent briefing can be found in [WCS02]. Some understanding of distributed systems and predicates in distributed systems has evolved as a result of the effort. Currently a distributed systems developer has few semantic handles and little tool support for finding faults causing most notorious software failures. This makes it a field with room for experiments and balancing conflicting requirements, which represents the core challenges of distributed computing.

The enormous complexity and non-determinism involved stands as a hurdle in adoption of distributed computing models. They are inherently more powerful then non-distributed models in terms of ability to provide quality attributes in solutions. There have been attempts to model execution of distributed software using relational and event-based behaviors [Bat95, Kun97, Sno88]. These models

formalize some intuitive observations of distributed systems, which are abstract in nature and require considerable effort and expertise to map to existing practices. As a result, they are not adopted in general practice. Event based models make some important assumptions. Events are generated and received by active agents. Every agent can be seen as a black-box with a monitor attached. The monitor is responsible for notification of events to agents which want to receive events. This results in clear division between the computation activity and the communication activity of an execution.

1.3   Complications involved

Debugging requires a thorough understanding of the system being debugged as well as the platform on which it is built. It is one of the processes which must directly deal with all details of distributed computing platform. There is no surprise in fact that if debugging is considered an art, process of distributed debugging is considered no more science than that.

1.3.1 Element of non-determinism

Non-determinism in distributed message passing program has been studied before [DKF93]. In treatise of effects of non-determinism on distributed development tools, there are two models of interest. First being the model of distributed execution itself and second, the specification of the execution. It is worth noting that specification of the distributed system affects the possible patterns of distributed execution, as specification is usually quite smaller then the execution. For a deterministic execution the specification along with initial conditions can be seen as an encoding of specific execution. In non-deterministic systems, there can exist many possible executions which are produced from the specifications and the initial conditions under control. The constraints on programming language limits possible execution patterns in software. This can be used for advantage by the development tool if constraints on original programming language are known or the programmer explicitly declares them to be followed by the specification.

1.4 Goals

It is required to find semantic handles for distributed debugging. How to have semantic handles which are not full abstractions of distributed nature of the system is the question in the picture. This trade-off allows simplification of debugging activity for domains which cannot afford performance hit of full abstractions.

Further, we need a framework for testing protocols for distributed debugging. which provides a readily usable tool in full-fledged development environment. Using such a framework new protocols for debugging can be tested.

2 Overview and Related Work

2.1 Model for distributed execution

A model for distributed computation platform is characterized by the assumptions about topology, the communication protocol and the assumed guarantees of the nodes it is built on.

In context of distributed executions, topology describes the configuration of communication channels between the nodes of the distributed system. Protocols are important devices for functional abstractions and building quality guarantees in the model of system which are not inherent in the communication infrastructure. The assumptions about individual nodes that make up the distributed system guides the complexity of the solution. These assumptions capture aspects such as maximum variation in delivery time of message and clock skew. The degree of variation between characteristics of these nodes is usually bounded in practice, but modeling them in theory is not easy.

Many guarantees which are taken for granted in sequential systems do not hold for distributed system models. The non-determinism associated with measuring the state of system as a whole in any generic asynchronous distributed model has motivated study of approaches that abstract out the non-determinism involved but the assumptions made about the model are either too simple or do not have clear reflections in the practice.

2.1.1   Sequential execution model

Sequential execution is a special case of distributed executions, in which there is only one node in the system. Sequential execution is modeled using notions of states and events. State is an abstraction of the observable behavior of system at any instance of time from specific location of reference. An external event is the stimulus for change in the state. An internal event is the result of change of state. An execution is viewed as a sequence of alternating states and events.

As it is easiest to see microprocessor as a device which takes an operator and produces result using specified operands, a simple representation comes out as n-tuple having exactly one instruction and n-1 operands.

Executions can take infinite time but the the algorithms used as the specification are finite in length. By specification of execution, we mean the machine-readable script describing the algorithm to represent all possible execution sequences. This can be achieved by allowing operations on the model of programming itself and introducing awareness of the programming language. Examples of such instructions are those which allow the execution to continue from an instruction which is specified as an operand. It is worth a note that this assumes a unique identifier associated with all instructions in the algorithm specification.

As model described above is very generic, specification standards usually support notions of reuse of specification in order to allow structures over the basic atomic specification primitives. These facilities do not increase the power of underlying model, but they allow increase in the productivity for the process of specification.

Such abstractions usually provide reuse of specifications or help imposing some desired constraints on the languages. Simple examples of such structures are C structures for specifying a multi-dimension assembly of variables. Another example is subroutines, which allow grouping instructions for reuse. Usually implicit or explicit name-

spaces are used in conjunction to impose desirable constraints and use these structures as proper abstractions.

2.1.2   Distributed execution model

Modeling distributed systems introduces an entirely new set of problems, which are non-existent in sequential execution models. In a generic distributed computation setting there are many compute nodes which are standalone systems by themselves. These systems solve problems which require collaboration by passing messages to each other. In pure distributed setting, we assume no shared resources between the nodes except the communication infrastructure, which does not have persistence attribute.

Events can be observed within the scope of the event. For distributed systems, there are two scopes to take into account. First is the local scope of the computation which is concerned with resources within the local address space of the machine. Second is the system scope, in which addresses identify the nodes. As these two scopes are usually isolated, we can abstract out the computation details of local scope and consider them as a series of sequential executions, when each sequential execution is defined between two system scope events.

Such an approach is usually taken to simplify the model of study, but it should be noted that the abstraction of the processes most of the times loses information which can be used for improving algorithms dealing with the system scope. However, most of the time volume of such detailed information is so large that it renders information useless in context of system scope. This is one of the reasons along with the conceptual simplicity that biases distributed system design to use hierarchical or simple and regular topologies. Due to the complexities of distributed systems few programming languages are available which have structures to abstract out the notion of sequential execution. Functional languages and dataflow languages provide good structures for capturing these interesting details efficiently, but imperative languages do not fall in this category. This is partly due to the power versus usability trade-off, partly due to the fact that imperative programming languages are usually not designed for such partitioning and partly due to lack of maturity in the methods to capture the characteristics of distributed system.

A way to look at the distributed system is to see all the operations as set of tuples as described above and then add to it operators for execution, which define the control flow of the execution. The essential

operators for theoretical models are described in literature [HK90]. As distance and time are usually not part of the expected results, we believe that programming language should allow constructs to capture these aspects without actually binding them with specific values. We describe these operators in brief as follows:

Sequence operator

Let t_1, and t_2 be instruction tuples defined as described above. A meta-instruction *(sequence, t_1, t_2)* is define, which implies that a valid execution should guarantee that t_1 will be executed before execution of t_2. It should be noted that in semantics of some imperative programming languages, this operator is implicitly associated with the location of instruction in the program, which means that it is the default operator in case that no other operator is specified explicitly. This behavior is problematic as it captures even those sequence relations which are not meant by the programmer. In other words, the default assumption is to introduce dependencies even when they are non-existent in the semantics of actual problem domain.

Choice operator

Let t_1, and t_2 be instruction tuples. A meta-instruction *(choice, t_1, t_2)* is defined, which implies that the execution sequence will continue either with t_1 or t_2 depending on the sample of the system state. Let *(choice, t_1, t_2)* be named as t_3. If t_3 follows t_1 or t_2 in sequence, then it is possible to generate infinite length execution from finite length specifications.

Concurrent operator

Let t_1, and t_2 be instruction tuples. A meta-instruction *(concurrent, t_1, t_2)* is defined, which implies that t_1 and t_2 can be executed simultaneously.

For ease of specification, these meta-instructions may take abstract groups of specification as operands. This also allows explicit definition of program structure. This is highly desired for capturing information that can be further used by development tools to ease troubleshooting.

2.1.3 Additional issues

To understand the additional issues when dealing with distributed system, we need to understand the effect of distance and the effect of time [Fid96] , which are mutually independent and independent to the effect of locations. By effect of distance, we mean the difference in the observations of the whole system from different nodes . An example of same is cross-communication patterns that we describe later. The effect of time refers to the non-determinism introduced by lack of global clock and non-deterministic delays. This reflects the fact that unique order on distributed events is partial order in contrast to the unique total order associated with sequential execution. Both the added aspects introduce non-determinism which cannot be modeled without extending the set of primitives for sequential execution. The effect of location is associated with local address space. Distributed systems must explicitly handle problems spanning these issues.

The first two operators described above are present in sequential executions also. The third operator is indeed new to distributed, and in general concurrent systems. Introduction of the concurrent operator breaks association of any point of execution with a unique point of time. As a consequence, multiplicity of the execution must be explicitly handled in order to make specifications unambiguous.

Today most of the programmers are familiar with using first two operators on abstract structures of specifications, the concurrency operator is used in its primitive form only. In other words, imperative languages do not provide language constructs for modeling generic concurrency in specification explicitly.

2.1.4 Message passing

A programming paradigm known as message passing programming has emerged from this scenario in which programmers explicitly use message passing primitives for sending messages and synchronization. Also these messages received are treated as external events by each node in the distributed system. As messages passing instructions send and receive are primitive instructions, it is hard to capture the structure of concurrency with them. This causes problems

for the tools that analyze the software for debugging. Message passing is usually implemented as libraries such as popular Message Passing Interface (MPI) [For93] implementations, which further weakens the association of concurrency specification with the core programming language. This is one of the reasons why utility of higher level semantic handles provided by libraries like MPI are not complete. Group communications is a good example of such a facility, which provides abstraction like broadcast, gather and scatter operations.

Software using message passing programming is designed in either symmetric or asymmetric topology. Asymmetric topology has some nodes in control of the system and others following decisions taken by these nodes. Symmetric topology allows all nodes to be treated as equals, but according to nature of application they are usually arranged in simple patterns like ring, grid, torus. In any sophisticated application, different logical topologies can be used for different purposes.

2.2 Debugging

The process of debugging is part of development and maintenance activities. There are two sources of information that debugger can

have. First and most important is the developer who specifies additional constraints to the debugger and controls the execution. Second is the semantic information in the program binary. This information can be used to detect runtime violation of these semantics.

2.2.1   Program translation

During process of compilation, a program is translated in the machine language. A program can be seen as an expression which imposes constraints on which paths software can take during the execution out of all paths that can be generated from arbitrary combinations of instructions. Some of these constraints are imposed by the semantics and constraints associated with the programming language and other are imposed by the programmer to model the domain of software application.

Even though software engineering attempts to establish standards for development processes in order to obtain quality of the solution, number of faults in software solution and the seriousness of their side-effects is far greater then other well-established engineering disciplines.

2.2.2 Tools to assist software maintenance

As a consequence of requirements of tools to assist software maintenance, it is important to have means of detecting, diagnosing and removing faults from software. To serve the purpose, three genres of software tools have emerged.

Static analysis tools use heuristics to discover patterns in specifications which are likely to cause anomaly even if it is syntactically correct. Testing tools take black-box approach and the software is verified for producing expected results for given set of inputs. The debugger is a tool to assist programmer to find fault manually, in contrast to other tools, which do not require involvement of developer to perform their function.

Static analysis tools do not have access to actual execution of software and limit their analysis to the information available from source code. Testing tools do not have access to internal view of execution, hence they can only help in automated detection of pre-specified assertions.

When a fault is detected in the software, it is necessary to understand and differentiate expected and faulty behavior of

application. This process requires introducing new constraints or assertions in the system from observations. Debugging is usually a cyclic process in which the cycle of observation, constraint insertion and scope reduction is followed iteratively until the scope of specification suspected for causing anomaly is reduced to a level where it is possible to deterministically state the cause of difference between the expected and faulty behavior. The process of correction then follows, which also involves many software engineering issues for modifying existing specification.

The debugging tools must solve the problems which are not detected by automated analysis tools. This makes debugging an art rather then process due to the fact that it has to deal with the systematic runtime analysis of the execution. However, the debugging process can be greatly benefited by having proper tools to capture frequently needed tasks. It is also important to have debugging support at language specification level and platform level. Debugging tools generally use additional information available from analysis of source code. This could be on demand or generated by a tool that analyzed specification before, like compiler.

Debugging can be on-line debugging or off-line debugging. In off-line debugging, information about the execution is collected but it is not analyzed simultaneously. In contrast, on-line debugging analyzes the information in parallel to the execution.

On-line debugging is further classified in interactive debugging and non-interactive debugging. Non-interactive debugging allows injection of additional constraints only before control flow is transferred to the initial instructions of execution. Interactive debugging, on other hand allows insertion of constraints when execution is active, requiring more sophisticated integration with programming and runtime platforms. Naturally, interactive debugging is easier to use for developer in detecting software faults.

2.3 Ordering distributed events

Distributed message passing executions are usually modeled as a partial order of events in system. Natural topology of system is then modeled using task-channel model, in which the execution nodes are assumed to be connected using channel entities. The partial order is defined by assuming that each process has a total order defined on its events at local scope. At distributed system scope, send and receive

events are considered. Further it is defined that if a message has a send event as e_1 and corresponding receive event e_2, then e_1 precedes e_2 (or e_1 "happened before" [Lam78] e_2) in the partial order. This definition follows from the natural observation that having isolated address spaces, causality relation can only be introduced by explicit message passing.

More work related to ordering of events in distributed systems can be found in [Fid91]

2.4   Detection of distributed predicates

According to definition of order on distributed message passing processes, a message receive can only take place on a channel of message send is performed on the same channel. This observation leads to notion of consistent cut, which is formal way of stating that at any observable point in the system, the number of sent messages on a channel is greater then or equal to number of received messages. The idea of consistent cut [CL85] was originally applied to checkpoint algorithms, which also deal with detecting consistent states of distributed system. The concerns of checkpoint algorithm are little more as it is also required to access all of the state elements as they

were before execution can proceed again. A predicate detection in general does not need to access all state elements after specific predicate is detected.

It is observed that set of all consistent cuts in a execution trace of distributed system forms a finite distributive lattice [GM01]. More work on distributed predicate detection can be found in [CG98, DB96, FZ90, MC88]

2.5 Debugging tools

Various debugging tools are designed after different debugging practices. Efforts to establish a standard for debugging primitives like high performance debugging standard [For98] have not been a full success. As there are too many debuggers described in literature to cover here, we list some debugging tools here which are more relevant to this work.

2.5.1 P2D2

AIMS project at NASA is building a debugger for parallel and distributed programs named Portable Parallel / Distributed Debugger (P2D2)[HJ00, Hoo96] . It uses open software gdb as the back-end for

पूर्णोत्तम गोविन्दराय केलकर पुस्तकालय
भारतीय प्रौद्योगिकी संस्थान कानपुर
ग्रन्थांक क्र० A...153069

debugging programs. It uses client-server architecture to enable heterogeneous debugging. It supports up to 128 processes running in a cluster environment. It has been used for debugging heterogeneous processes running under Globus.

2.5.2   CDB

CDB [WCS20] is a debugger developed for cluster applications. It is a tool to debug in heterogeneous environment and is based on Java. It also allows replay of recorded executions.

2.5.3   TotalView

TotalView [Etn] is commercial tool for MPI and OpenMP debugging. It provides tools to visualize the information effectively. It supports MPI debugging features like message queue visualization.

## 2.5.4   IBM Distributed Debugger

IBM distributed debugger [MMP+96] attempts to abstract out the distributed nature of computation by providing distributed extensions to sequential concepts like distributed stack, remote procedure step into.

3 Effect of Topology and Protocols

A generic distributed computing model does not make many assumptions about topology and protocols of the system in practice. Many problems that are not solvable for general setup can be solved by defining topology or protocol.

We use simple graph figure to show the topology and sequence diagrams of protocols. In sequence diagrams vertical lines are the process time lines. The slant lines are messages. The higher end of slant line is send event and the lower end is receive event.

In early attempts to abstract out distributed nature of systems, remote procedure call was used, in which process waits for confirmation of receipt, optionally with result message for each message sent. This protocol attempts to use client-server model for each procedure call in order to mimic a synchronous system.
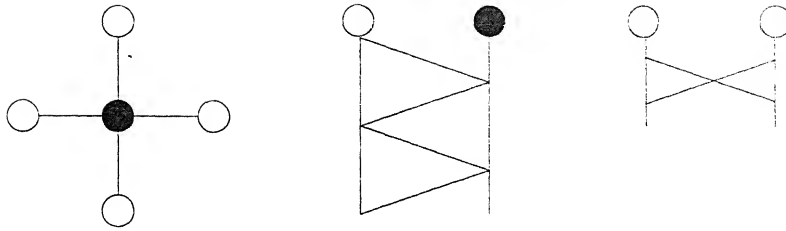
1. Star topology, RPC and cross communication patterns

Apart from being synchronous, the RPC mechanisms are also used as they are simple to analyze and debug due to their conceptual simplicity. Figure 1 shows the topology and communication pattern of protocol for such systems. One of the property of such a communication is the ability to detect the relative sequence of messages without attaching tags to the messages. This is due to absence of cross communication patterns depicted by Figure 1. Such patterns show the effect of distance in distributed systems. Two processes which emit events simultaneously from specific frame of reference observe those events in mutually inverted order. Such events are possible in asynchronous symmetric systems in which any point can emit an event.

By regulating cyclic structures at protocol or topology, non-determinism of a distributed system can be decreased. This patterns can be used by software analysis tools for further checking. Also, it can

reduce the overhead needed for recording the execution trace. For example, if two processes agree on explicit channel reversal protocol, that is to say that the communication is half duplex between the nodes, then cross communication cannot take place

The property that is imposed by the protocol and topology is the acyclic nature of the communication patterns. Such observations are equally useful in establishing structures on temporal dimension also. The half duplex communication pattern can be modeled using a token in the channel. A process having token can send messages. After sending messages for some time, it passes the token over to the other processes and thus explicitly switching the direction of channel. By semantics of such a protocol, now it cannot send any messages while other processes is sending the messages using the token.

In a more general setting, this can be extended to single speaker multi-listener setting. Only one of the communicating process can transmit at any time. This is modeled by using a token ring model (not to be confused with token ring LAN). A token is passed through the ring and the node having the token is allowed to send messages to others, while all others are not allowed to send the messages. For a generic setting of distributed system, the underlying physical topology

is not known and logical topology is a mesh in which each node is connected with all others.
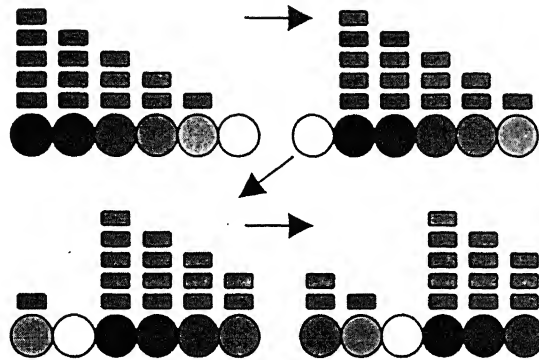
3.1 Token-mesh framework

We propose a framework for protocols which we call token-mesh that implements explicit channel reversal protocols over all its channels. Our base model of network has mesh logical topology and every channel has a token associated with it. The token is passed by a message, explicit or piggybacked, from one node to another. Node having the token can send messages over the channel. We assume that underlying communication infrastructure handles reliable delivery. Figure 2 shows token ring and token mesh topologies. In token ring topology, the dark node has ability to send messages to others as it has the token. This token is passed to next node in ring after some time. In token-mesh topology, there are multiple nodes sending information over different channels.
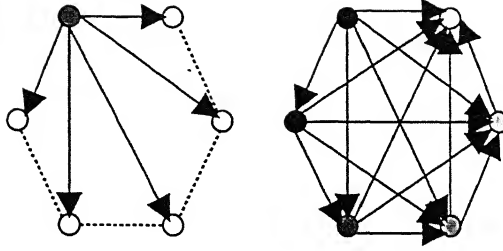
2. Single token and multi token mesh

We describe a protocol built on this infrastructure as an example. It is inspired by the HPC scientific applications designed using SPMD paradigm with message passing, where the applications start with a fixed number of nodes and the response of application does not have real-time constraints. This protocol works below the application level message passing transparently and can provide an interface for tracing to application. Nodes in the given configuration are arranged in a total order. Assume there are N nodes in configuration. To begin with the first node is assigned N-1 tokens, second N-2 tokens and so on. The last node does not receive any tokens. According to the protocol, nodes can send messages on the channels for which they have token. This setup is shown is Figure 3. After predefined time interval, the first processes releases its tokens to all other nodes. This results in all other nodes receiving one more token than what they had. This token passing goes

on in a round robin fashion so that all processes get equal chance to use the channel in long term.


3. Sample protocol in token mesh framework

At the end of each round, there are N messages (or one broadcast operation) for synchronization. These messages explicitly mark end of a round and also act as a point of reference for determining direction of channel. This results in reduction in amount of information to be passed around to determine the sequence of events at higher levels. As it can be seen, it is not possible to have any cyclic message paths between explicit synchronizations. Purpose of this example is to instantiate token-mesh framework. We do not analyze fault-tolerance of this protocol and modifications to improve the same. Within the token-mesh framework, various protocols can be instantiated by altering ways of assignments of tokens.

3.2 Execution patterns and reducibility

Viewing distributed execution trace from point of view of consistent cuts provides interesting observations. In general, the size of lattice of consistent cuts is bounded by the number of events in the system. As message passing primitives are used in non-structured forms, there is no simple handle on the analysis of such execution traces. One particular observation we make is that lattices are recursively substitutable. If we take an element of lattice and substitute it by placing it such that the upper cover elements of original element now become upper cover of greatest element of substituted lattice and the lower covering elements of original element become lower covering elements of least element, the resulting partial order is again a lattice. This fact can be used to define higher order concurrency constructs in term of primitives. The fork-join constructs of concurrency provide a base of structured concurrency. Figure 4 illustrates the recursive use of fork-join constructs in one of the forked processes which allows building new lattice by substitution. The greatest element is a fork and least element is a join. Expanded figure illustrates substitution of same pattern for one of the process events.
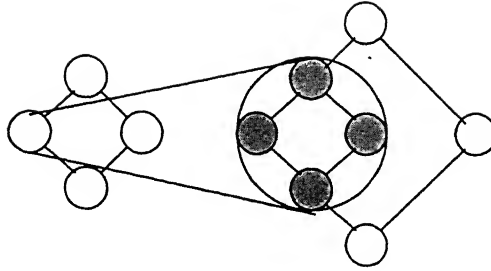
4. Lattice substitution

Having such semantics in distributed programming languages is beneficial. Hierarchical structure created by such a model is more powerful then the flat process model for distributed computation used message passing practices.

3.3   Limiting possible states.

The complexity of messages passing is limited by use of synchronization primitives. As synchronization is costly in general asynchronous setup, combining coarse grain synchronization primitives with message passing offers a good handle on simplifying the model of system.

Let us consider a space in which time-line of each node represents a dimension. Asynchronous message passed from one dimension to other binds the dimensions so that the number of states for the

combined subspace formed by these dimensions is the product of individual number of states. On the other hand, synchronization primitives unbind these dimensions as execution must pass through the point in subspace defined by the synchronization. This means that total number of states in subspace having a synchronization is sum of states before the synchronization and states after synchronization. In other words, by having bounds on the number of events between synchronization, we can produce a bound on the width of execution lattice, which gives one handle on commenting about complexity of distributed system.

To describe complexity of any message passing application, it is interesting to see that from how many nodes a node can receive message without going through synchronization with all of them. For example, client-server communication patterns receives synchronization on all messages. The example protocol described goes through synchronization after specific time. Between two synchronization, the communication patterns are limited to acyclic patterns in order to forbid cross communications.

As discussed before, the longest chain in a lattice is bound by number of events. A handle on easy to analyze trace can be described

as having a topology and a protocol such that the execution trace is recursively reducible and placing synchronization and message passing primitives in a manner which allows placing bounds on the width on the lattices of consistent cuts. This can be related to the bounds on number of events between synchronizations. Intuitively it can be seen by considering two special cases. In case of a total order between two communicating nodes, the width will be 1. If these are unsynchronized concurrent events, then the width becomes 2. The example can be extended for more number of events in various configurations.

4 A Testing Platform

For study of distributed debugging protocols, we created a tool in which protocols can be plugged in to experiment with. The purpose was to establish a framework which can be re-used for implementing various protocols and to implement protocols which can be used for debugging symmetric distributed components.

4.1 Architecture

The tool is built for debugging collection of components hosted on different physical nodes. A component is a piece of software that is only meaningful in context of a container. The container provides an environment and services that component assumes for its correct function. Typically container also provides life-cycle management services to the component. Lately one of the computing models that has gained popularity is one in which source code is compiled in platform independent intermediate code and then this code is run inside

software virtual machine. This pattern is closely comparable with a component running in container.

The advantage of such a computing model is ease of handling heterogeneous environment and portability, which are important issues in deployment of distributed software. Figure 5 illustrates the architecture.
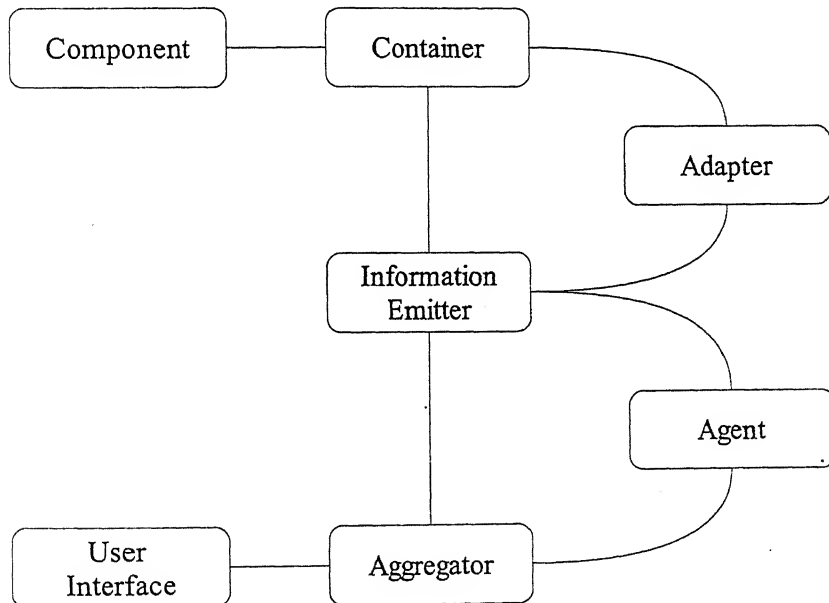
5. Architecture for debugging components

4.1.1 User interface

The user interface provides graphical and textual interaction to issue command for debugging. The commands can be control

commands or monitoring commands. User interface is also responsible for presenting information gathered from Aggregator in comprehensible format.

4.1.2 Aggregator

Aggregator is responsible for communicating with the information emitters for components and agents. It's responsibility is to provide protocols for communication with the information emitter.

4.1.3 Agents

Agents are optional. They can encapsulate various aspects of system like providing a hierarchical structure of aggregators by acting as an aggregator, enforcing security policies in grid environment and so on.

4.1.4 Information emitter

Information emitters publish their network address to which the aggregators or agents can attach. They also support the protocols used by aggregator.

4.1.5 Adapter

An adapter is optional and is useful in cases that some type of protocol conversion is required or some feature is to be implemented on the side of target process. An adapter can also be a container / wrapper for software which is not hosted inside a container. For example, it can be a full fledged debugger encapsulating an execution.

4.1.6 Container

Container communicates with the information emitter to carry out the debugging commands on the components. It observes component for a set of events and reports its status as requested.

4.1.7 Component

The component is the target of debugging. They can be classified in read-only and controllable components. A read-only component can model an interface between other two components. An example would be observing HTTP requests over the network by a monitoring component. Controllable components can allow varying degree of control over their executions.

We list commands that should be implemented by the components.

- Attach: Establish communication with debugger.

- Detach: Close communication with debugger.

- Unconditional Stop: On receiving this command, the process should halt its execution in next possible state.

- Unconditional Resume: On receiving this command, process should immediately continue the execution.

- Conditional Stop: A process should wait until a condition becomes true. When it does, it should stop execution.

- Reset condition: Specified condition is removed from set of conditions on which the process should stop.

Apart from these control commands, the component can provide a way of evaluating an expression. The expression is passed as a string by user and the intermediate mechanism need not know semantics of the same.

4.2 Deployment platform

The debugger is tested on a cluster of Sun UltraSparc machines. Solaris 9 operating system was installed on these machines to make a cluster. The debugger itself was running on a Linux machine. Lam

MPI was installed as message passing library. The mpiJava library was installed on top of the Lam MPI to enable message passing programs using Java. MPI was chosen over RMI for low-overhead message passing

4.3 Technologies used

4.3.1 Programming language

Java was obvious choice for the debugger due to many reasons. Java is well known for portability of its programs. This is required for developing platform independent debugger which can be deployed anywhere. Java also provides Java Platform Debugger Architecture (JPDA), which partially fits in the model that we described. Software written in Java runs under a virtual machine, which provides desired facilities for interactive debugging

4.3.2 Client platform

To implement the tools, it was necessary to make it extensible, platform independent and modular and at the same time provide easy to use and conceptually simple interface, which tools of today lack. We

chose NetBeans as the implementation platform for following reasons. NetBeans provides rich interface building concepts in the API, which allow building tools that adopt the changes with time. It is widely used IDE so adding the tool to it promptly enables rich features of the IDE to be available to the user of debugger.

However, due to major version change in the IDE, the documentation at point of time is not comprehensive and learning curve is quite steep as the model of extending IDE is far from simple and straightforward. However, the effort is justified by having benefit of full fledged IDE.

The module has been developed to keep the dependencies limited to the core subset of NetBeans platform, which does not provide all development tools that IDE has built in. It can run using lesser system resources compared to NetBeans IDE. This reduces the demands on the system to run the tool.

4.4 Protocols

To instantiate the aggregator, we implemented some protocols for halting which are meaningful for SPMD applications. The interface is mainly concerned with two entities and their interplay. One is process

group. A process group is the set of processes on which identical commands might have been issued when debugged separately. The other notion is of conditions, which instruct the processes to stop when it becomes true. For sake of simplicity, we implemented line number conditions for the debugger.

The first protocol called SimpleAggregator simply relays commands to individual target processes. It is possible to exercise all primitives listed for component above with it.

We tested the tool with these protocols in the deployment platform setup. It worked smoothly for our target programs that were written using mpiJava. It is particularly effective for programs that has some symmetry which can be used by proper process grouping.
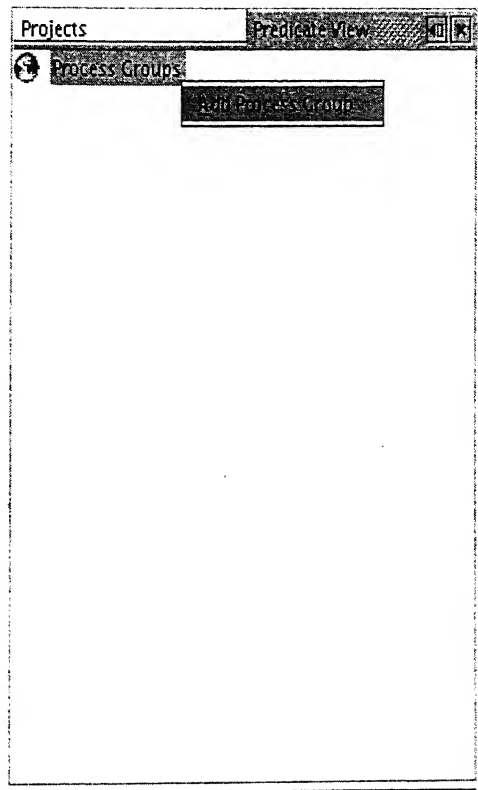
4.5 Debugging activity

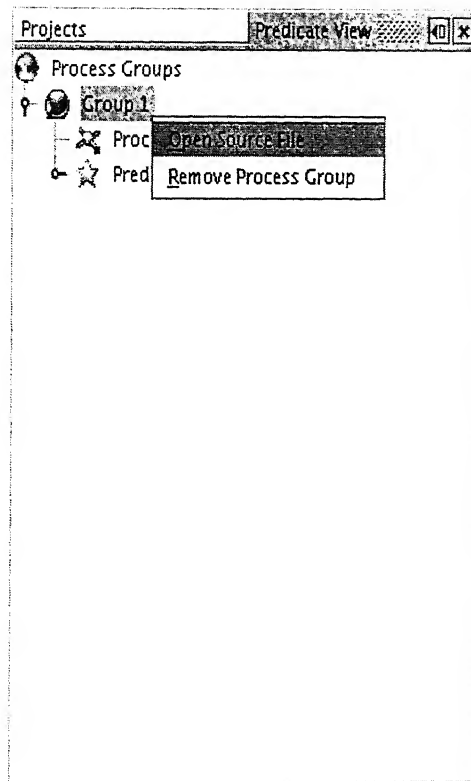The debugging activity is illustrated graphically in following screen-shots.
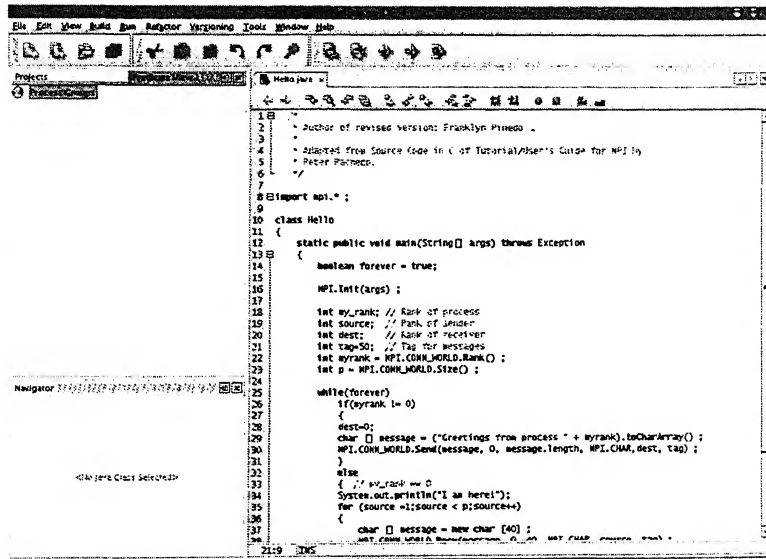
6. IDE window

The "Predicate View" frame can opened from windows menu. After that it appears like Figure 6.
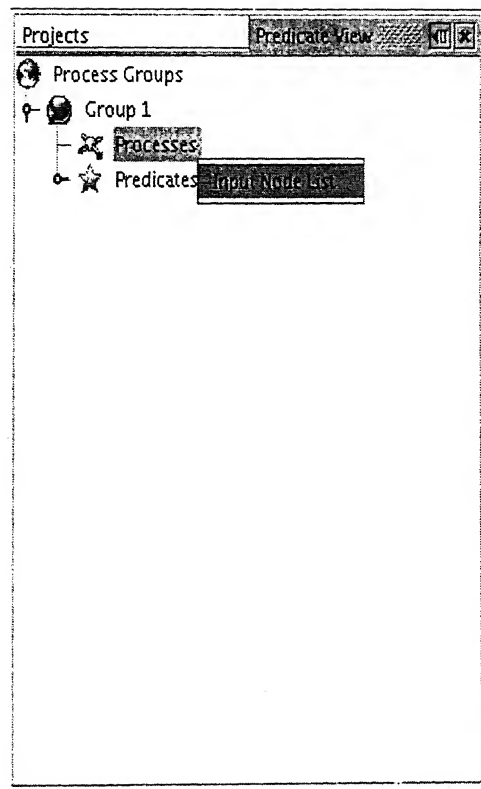
7. Process group creation

In "Predicate View" frame, the process groups can be added using the context menu as shown in Figure 7.
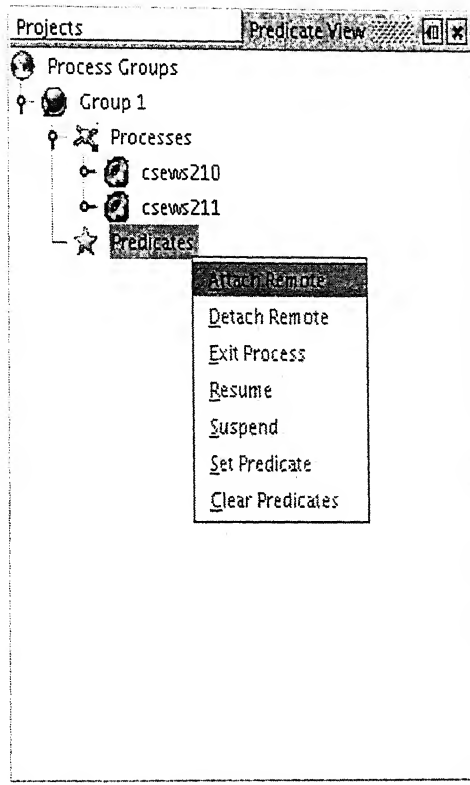
8. Process group actions

A process group can be used to open a file or the group can be removed using the context menu as shown in Figure 8.

9. Adding nodes

The Processes entry allows adding nodes to a group. Figure 9 illustrates this.

10.Predicate actions

From the predicates entry, various debugging actions on the group can be used. Actions not available are shown in faded color in context menu. Figure 10 illustrates the same.

5 Conclusion and Future Work

We show that it is possible to use semantic handles for simplification of distributed debugging process. The design of a distributed programming language should introduce better structures over concurrency operations which not only prevent bugs at compilation phase, but also can be utilized by software analysis tools like debugger to detect anomalous behavior at runtime. We showed that token-mesh framework can be used to instantiate protocols which use explicit channel reversal. We pointed out that good semantic handles restrict the width of the lattice of consistent cuts.

The debugging infrastructure can be used to test different protocols for various classes of distributed execution models and to verify their utility on actual applications.

In future, the order on the classes of distributed executions from point of view of abstraction can be studied. The tool can be extended with more sophisticated protocols for debugging.

References

[Bat95] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. Comput. Syst.*, 13(1):1-31, 1995.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63-75, 1985.

[CG98] Vijay K. Garg Craig M. Chase. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191-201, 1998.

[DKF93] Suresh K. Damodaran-Kamal and Joan M. Francioni. Nondeterminancy: testing and debugging in message passing parallel programs. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 118-128, New York, NY, USA, 1993. ACM Press.

[DB96] Lucia Maria de A. Drummond and Valmir C. Barbosa. Distributed breakpoint detection in message-passing programs. *J. Parallel Distrib. Comput.*, 39(2):153-167, 1996.

[Etn] Etnus. Totalview. "http://www.etnus.com/TotalView/".

[Fid91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28-33, 1991.

[Fid96] Colin Fidge. Fundamentals of distributed system observation. *IEEE Softw.*, 13(6):77-83, 1996.

[For93] The MPI Forum. Mpi: a message passing interface. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 878-883, New York, NY, USA, 1993. ACM Press.

[For98] High Performance Debugging Forum. High performance debugging version 1 standerd. "`http://ptools.org/hpdf/draft/`", 1998.

[FZ90] Jerry Fowler and Willy Zwaenepoel. Causal distributed breakpoints. In *Proceedings of Tenth International Conference on Distributed Computing Systems*, pages 134-141, Paris, France, May 1990.

[GM01] Vijay K. Garg and Neeraj Mittal. On slicing a distributed computation. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 322, Washington, DC, USA, 2001. IEEE Computer Society.

[GVS94] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. *SIGPLAN Not.*, 29(9):140-148, 1994.

[HJ00] R. Hood and G. Jost. A debugger for computational grid applications. In *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 262-270, 2000.

[Hoo96] Robert Hood. The p2d2 project: building a portable distributed debugger. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 127-136, New York, NY, USA, 1996. ACM Press.

[HK90] W. Hseush and G. E. Kaiser. Modeling concurrency in parallel debugging. *SIGPLAN Not.*, 25(3):11-20, 1990.

[Kun97] Thomas Kunz. High-level views of distributed executions: Convex abstract events. *Automated Software Engg.*, 4(2):179-197, 1997.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558-565, 1978.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593-622, 1989.

[MMP+96] Michael S. Meier, Kevan L. Miller, Donald P. Pazel, Josyula R. Rao, and James R. Russell. Experiences with building distributed debuggers. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 70-79, New York, NY, USA, 1996. ACM Press.

[MC88] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 316-323, 1988.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Trans. Comput. Syst.*, 6(2):157-195, 1988.

[WCS02] Xingfu Wu, Qingping Chen, and Xian-He Sun. Design and development of a scalable distributed debugger for cluster computing. *Cluster Computing*, 5(4):365-375, 2002.